AI-Powered Bug Hunting - Evolution and benchmarking

Alfredo Ortega - ortegaalfredo@gmail.com
X: @ortegaalfredo
Neuroengine.ai

June 27, 2024

While AI holds promise for assisting with bug hunting, its actual impact remains unclear. This presentation addresses this doubt by introducing Crash-Benchmark, a standardized evaluation framework for AI-driven static analysis tools. We'll share results from a simple bug-hunting AI agent, AutoKaker, and discuss the implications for optimizing AI-based bug hunting in C/C++ code bases.

1 Introduction

Opinion on automatic bug finding is controversial. At the date of this article's publication, there is no consensus about whether this is possible or not, or to what extent. This is partly due to the rapid advancement of LLM models; up until months ago, open-source models were not advanced enough to be effective at bug finding. There exists a threshold in the complexity of LLMs beyond which bug finding becomes possible, and in this article we benchmark various models and found that this threshold has been reached for some vulnerabilities.

2 CrashBench

Crashbench [1] is a simple automatic test-case based benchmark tool. It connects to serveral LLM services offering opensource and private AI models, send a test case and then measure results. If the LLM found the bug in the correct line, then score is increased by one.

2.1 Design

Most of the test cases for the v1 version are based on Gera's Insecure Programming exercises [2], plus 3 real vulnerability examples. The LLM is assigned a score based on the number of vulnerabilities that were reported, with real vulnerabilities having 10 times the score.

The configuration of Crashbench is a single .ini file containing the prompt, test case files and expected lines where the bug is found.

```
[SETTINGS]
SystemPrompt="You are an expert security researcher,
              programmer and bug finder."
Prompt="Check this code for any out-of-bounds or
        integer-overflow vulnerability, explain it
        and point at the line with the problem,
        and nothing more, in this way:\n'Bugline=X'
        where X is the line number of the bug,
        and then print that line number.
        If the code has no bugs, then print 'Bugline=0'."

[Basic]
file1=stack1.c,6
file2=stack2.c,6
file3=stack3.c,6
file4=stack4.c,6
file5=stack5.c,6

[ABOs]
file1=abo1.c,4
...
```

In this way, the test not only tests bug finding capabilities, but also accuracy in reporting. Many models are good at finding vulnerabilities, but they fail at accurately pointing exactly where the bug is located in the code. To create negative tests (tests where no vulnerability should be detected), just set the expected vuln line number to zero.

2.2 Parameters

Software used was vllm v0.5.0.post1 [3] for AWQ quantization and aphrodite-engine v0.5.3 [4] for EXL2 quantization. Parameters used for inference using vllm were:

- temperature: 1.2
- top_p=1.0
- frequency_penalty=0.6
- presence_penalty=0.8

2.3 Results

The benchmark ran against 16 LLMs, most of them being the latest versions, but also some older models based on Llama-2 to compare them. Additionally, several quantizations of the same model were tested to measure the effect of quantization on LLM bug-reporting accuracy.

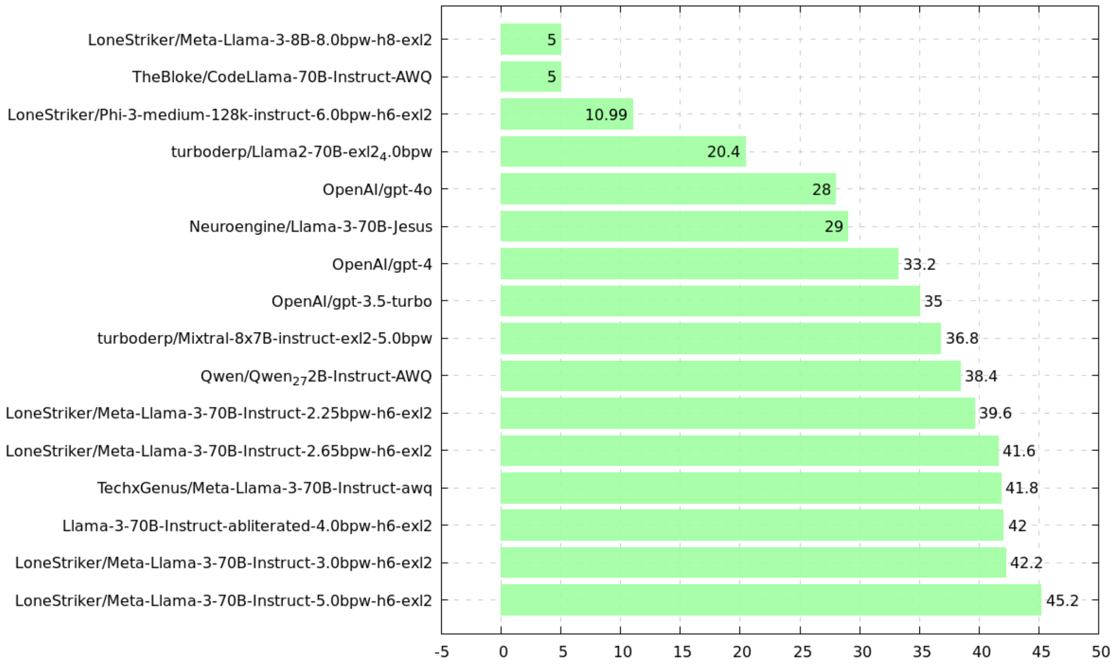


Figure 1: Crashbench score

As shown in Figure 1, Older models are not competitive at code understanding and bug finding, with newer models being significantly better. Even closed models like ChatGPT are surpassed by these newer models in terms of performance. Additionally, the relatively small effect of quantization on results is evident, as a strong quantization of Llama-3-70B (2.25 bpw) did not have a significant impact on the model's score.

2.4 Quantization effects

At Figure 2, we now focus on the effects of quantization on the score. Quantization is a technique that compresses models by representing weights using fewer bits, losing some quality but reducing the amount of memory needed. This results in increased speed and efficiency. Since current GPUs are mostly limited by memory bandwidth, the efficiency of inference decreases nearly linearly with size.

We set the y-axis to 0 so that it can be more easily seen how low an effect quantization had on the scores. We can also see the rapid increase in size with the increase of bits per word, without any corresponding increase in score.

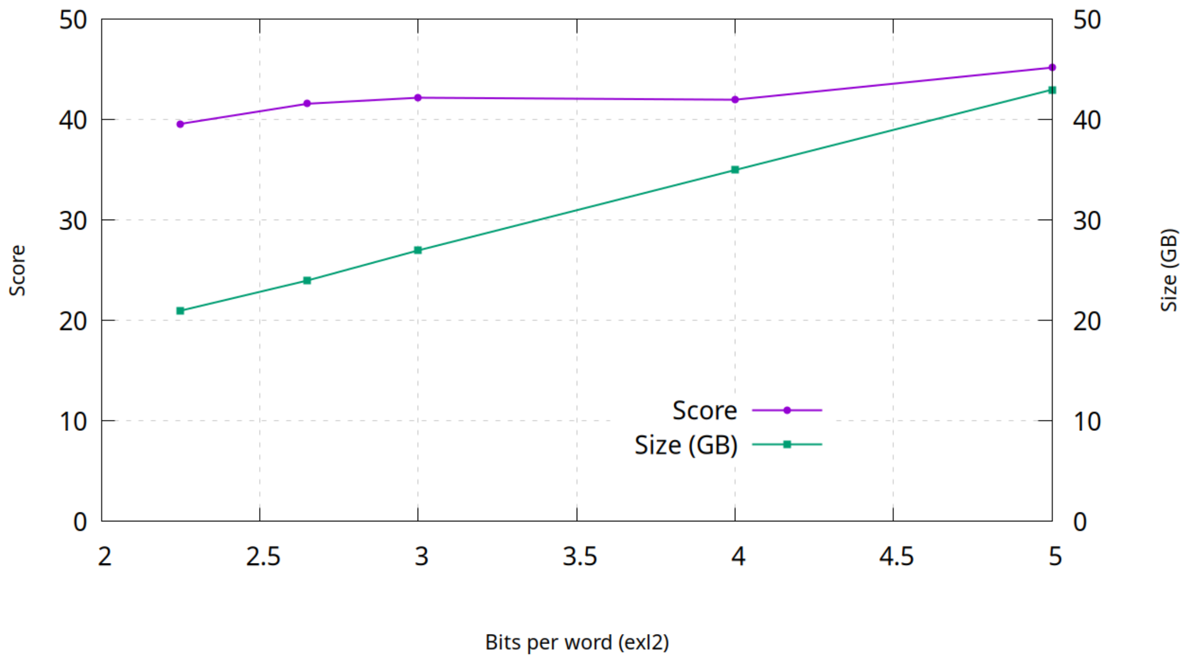


Figure 2: Quantization effects on score. Model: Meta-LLama-3-70B-Instruct.

We can plot a second graph at Fig. 3, showing efficiency of the different models, meaning the score per size in Gigabytes. With decreased size, speed and power required for inference also decreases linearly, increasing efficiency of operation.

We can see how the current most efficient models are highly quantized versions of Llama-3 70B. At around 25 GB, those models are still out of reach for most personal home computers. The best next option would be to use a highly quantized version of Mistral-8x7B, which can run on CPU on most modern computers at an acceptable speed.

2.5 Crashbench vs LMSys ELO

The LMsys leaderboard [5] has become the industry standard for model benchmarking. We can compare how our bug-finding benchmark correlates with the overall model score.

Intuitively we would assume that overall ELO and crashbench scores should be somewhat related. But in 4 we can see some inconsistencies, especially with modern OpenAI models. These models have much better ELO scores than Crashbench scores. This means that these models are much better as generic assistants and code generation than at bug finding. We suspect that super-alignment might cause these models to refuse to show bugs, as an analysis of gpt-4 and gpt-4o shows that they do not show many wrong bugs or lines on the test-cases; instead, their low scores are mostly due to denying that there is a bug at all. Low scores might also indicate problems on the benchmark, as we discuss in the following section.

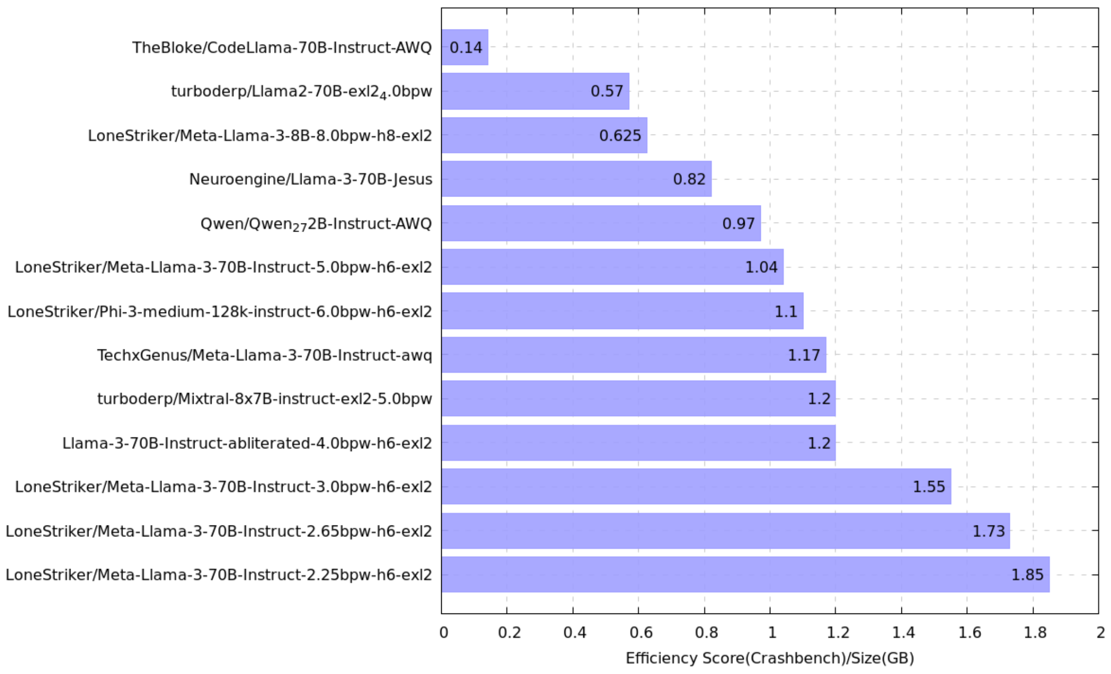


Figure 3: Total model efficiency. This graphic shows how many points the model have for every GB in size.

2.6 Problems

Problems that may affect this benchmark accuracy are:

Incorrect parameters and/or prompt format: Instruct models have a specific format that must be used on the prompts to maximize their understanding of the requests. Many LLMs are quite flexible on this format, while some are not. It's important to respect the prompt format of each LLM to maximize their code-understanding capacity.

Model trained on the solutions of the benchmark: As most models are trained on terabytes of tokens, it is very likely that the test cases, both artificial and real, were part of their training, along with the solutions. This might introduce a bias where models are very good at passing the benchmark, but not so good in real-world applications. The solution to this problem is to create more unpublished test cases that the LLM didn't see during training. However, this is a short-lived solution as it's very likely that newer versions of the LLMs will contain these new test cases, so they must be discarded in every new version of the benchmark.

Bugs on inference software/quantization quality: Inference software is evolving rapidly, and it contains bugs that affect quality and reasoning. A solution to this problem for benchmarking is to always use the same inference software. In our case, we use either vLLM or Aphrodite engine, which internally uses vLLM.

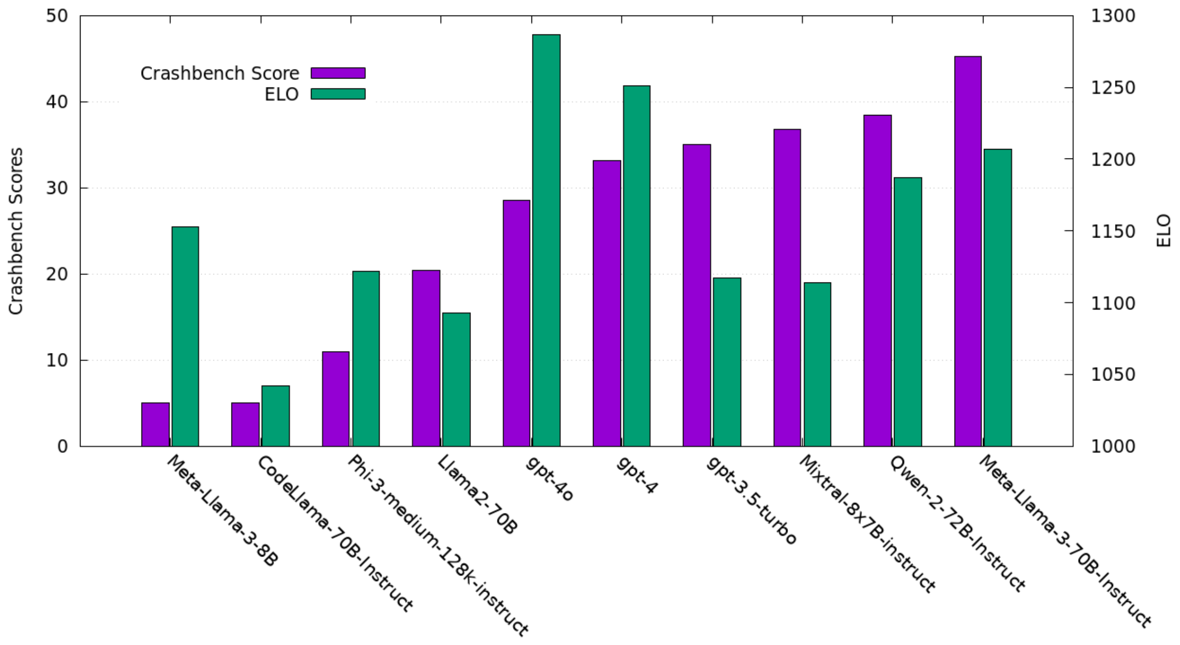


Figure 4: Crashbench score vs Overall model ELO score. We can see a general correlation except on closed models.

Refusals due to alignmnet: Some models refuse to discover bugs because they reason that they can be used for malicious purposes. This can be bypassed with several techniques such as prompt jailbreaking or abliteration, but both techniques might affect the code-understanding capacity of the model. However, the abliterated version of Llama-3-70B was compared against the original version and showed a minimal effect on the results.

3 AutoKaker: Automatic vulnerability discovery

Using the same technique of the benchmark we can easily construct a tool [6] that process source code and annotates every vulnerability found. The algoritm described in fig 5 is simple:

1. Separate source code into individual chunks that contain one or more functions
2. Assemble a prompt asking the LLM to analyze the code
3. Annotate the results

This tool (see fig 6) can be launched on complete codebases and will annotate every function with possible vulnerabilities, ready for triage and exploitation by a human operator. Unlike other approaches, this tool does not attempt to verify or exploit the

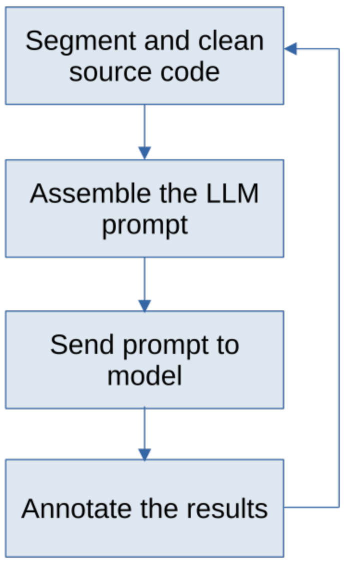


Figure 5: Autokaker main loop

vulnerabilities found, as this is a much more complex task. We propose in the next section that it is unnecessary. The tool currently supports only C code, but this is a limitation of the current code parser due to its inability to separate functions. The tool can run on C++/Rust code with a modified code parser.

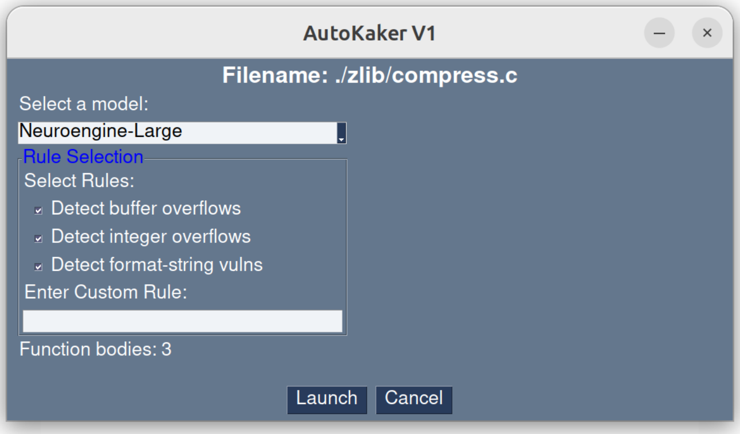
3.1 Problems with automated AI exploitation

We can see a simplified diagram of the stages of vulnerability discovery at 7. Once we found a possible vulnerability, we have two paths: Either confirm it via exploitation, or fix it via a patch. We can do two important observations:

- Is not necessary to confirm a possible vulnerability to patch it. This follow the philosophy of defensive programming.
- Patching a vulnerability requires much less skills than exploiting it, or even finding it.

Similar tools/benchmarks such as Meta's CybersecEval2 [7] and Google Project Zero Naptime [8] aim to find and verify vulnerabilities, and due to the high-skill and high-complexity nature of this task, current AI systems perform poorly at this. They can only succeed in basic examples without any software protections or exploit countermeasures.

While offensive AI will eventually become advanced enough to succeed at this task, due to the observation that it's often easier to fix a vulnerability than to create an


AutoKaker V1

Filename: ./zlib/compress.c

Select a model: Neuroengine-Large

Rule Selection

Select Rules:

- ✓ Detect buffer overflows
- ✓ Detect integer overflows
- ✓ Detect format-string vulns

Enter Custom Rule:

Function bodies: 3

Figure 6: AutoKaker GUI

exploit for it, we can assume that the asymmetry between defense and attack will cause offensive AI-generated exploits to almost never succeed. This is because less complex defensive AI will discover and patch them first.

Another conclusion is that since current LLMs are advanced enough to discover some vulnerabilities, they also have the capacity to automatically patch them, as shown in the next section

4 Auto-patching

Vulnerability discovery/annotation and vulnerability patching have similar workflow, but instead of adding a comment describing the vulnerability, we ask the LLM to generate and add code that fixes it. The autokaker tool can already perform this task by using the –patch command-line argument, displaying a simple GUI (see fig 8).

4.1 Iterative patching

Most SOTA LLMs like Llama-3, Mistral-Large, GPT4, Gemini or Claude are already capable of generating patches but they do not have a 100% rate of success. Meaning that the generated fixes will sometimes either not compile or create additional bugs.

We solve this problem using a closed-loop approach (see fig 9), in which after every patch generation, the autokaker agent checks if the code compiles and passes all tests. If the LLM code fails to pass these tests, we can try multiple times until the generated code passes all tests. Notably, most SOTA LLMs generate correct patches on the first try.

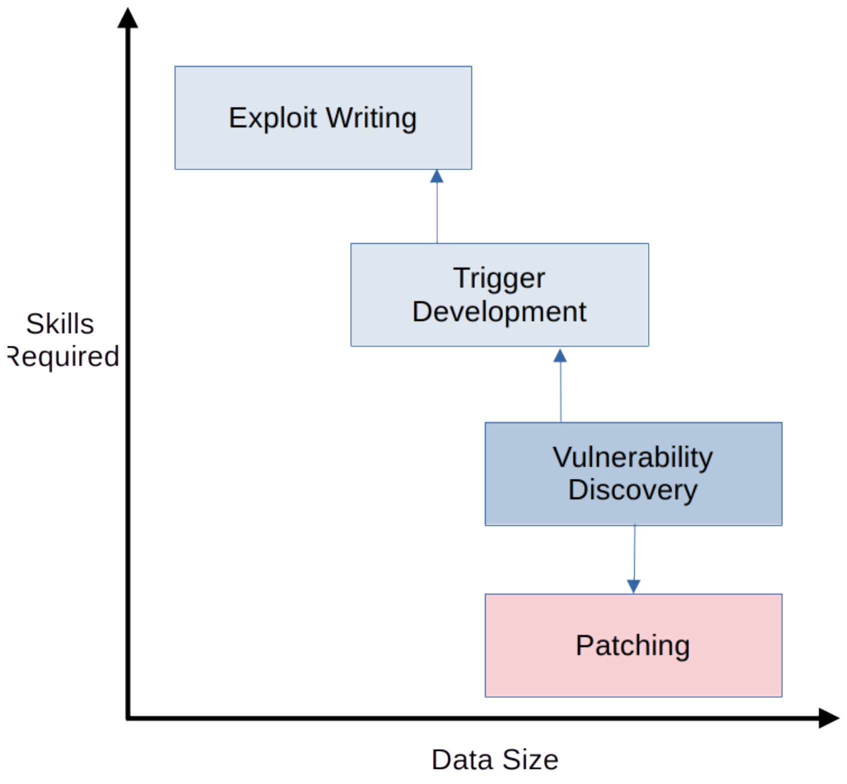


Figure 7: Simplified vuln discovery stages

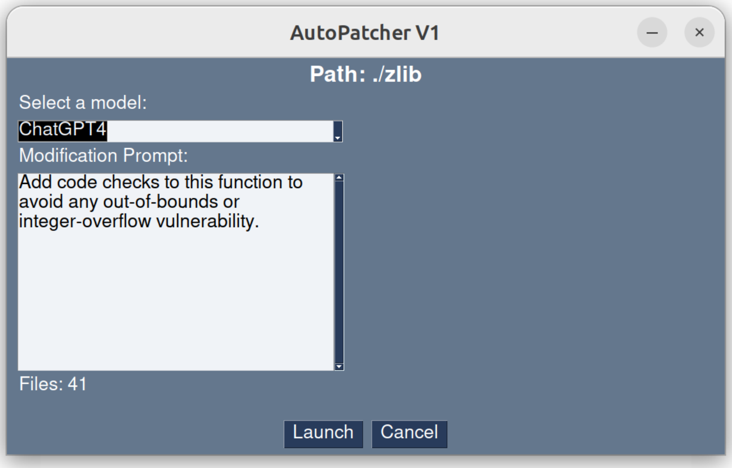
4.2 Example: zlib-hardcored

Zlib [9] is a compression library that is small, and include example utilities that compress/decompress binary data, that can be used as a test for the correct workings of the several algorithms implemented. The autopatcher utility was run on this code using this command line:

```
cd zlib;python autok.py --patch --make "make&&example64" .
```

This will run the autopatch recursively on all .c files and run the command 'make && example64' after each modification to check for the correctness and validity of every patch.

This generated a compatible refactor of the original zlib library with over 200 applied security patches. The hardened zlib code can be downloaded at [10]. Notably, the modification of this project to add additional checks was done 100% automatically



Select a model:
ChatGPT4
Modification Prompt:
Add code checks to this function to avoid any out-of-bounds or integer-overflow vulnerability.
Files: 41

Figure 8: Autopatcher GUI

with no human intervention. While not all patches fix exploitable vulnerabilities, they add defensive programming that protects the zlib function from many future unknown vulnerabilities, with the added benefit of randomizing the implementation itself, making ROP attacks much harder.

4.3 Example: OpenBSD-hardcored

Second example is the OpenBSD kernel. OpenBSD [12] is an operating system known for its security and correctness. However, the Autokaker tool discovered many vulnerabilities, making it a candidate for autopatching.

At this time, autopatcher was run on the complete netinet/netinet6 system using GPT4 as a model, generating around 2000 security checks [11]. Note that most patches will result in unused code, and most checks are not really needed, following the same philosophy as defensive programming.

As OpenBSD does not have tests that check the correctness of the IPv4/IPv6 stack, patching was 'blind' in the sense that they may generate errors. Therefore, the patches had to be reviewed manually. However, out of thousands of modifications, only 2 patches needed manual correction.

It is not recommended to use this 'hardened' code in production as it still might contain bugs introduced by the autopatcher and not yet detected. Also, as we discuss later, the patches can be easily regenerated with a newer, more powerful LLM.

4.4 cost

Currently, the complete refactor of the netinet/netinet6 subsystem of OpenBSD 7.5 is the biggest project that has been autopatched. We can cite some numbers of the associated cost:

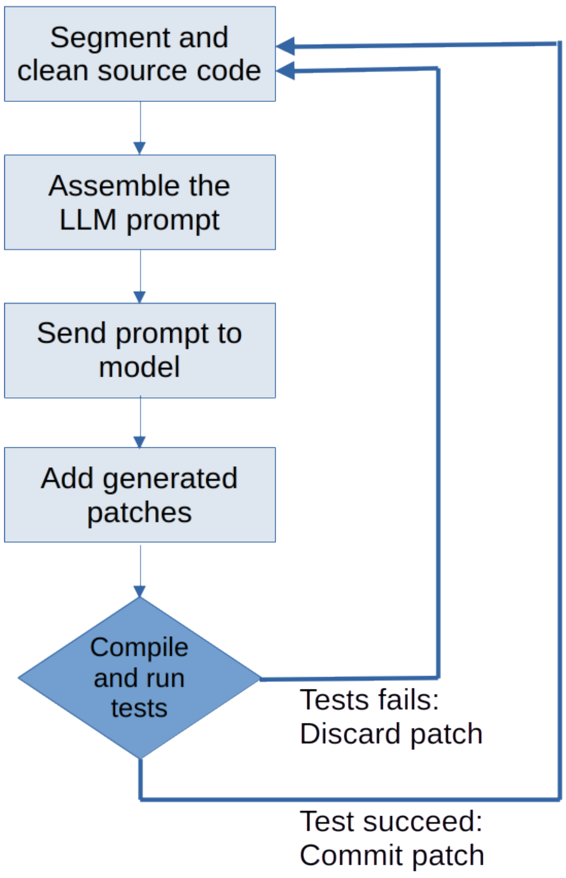


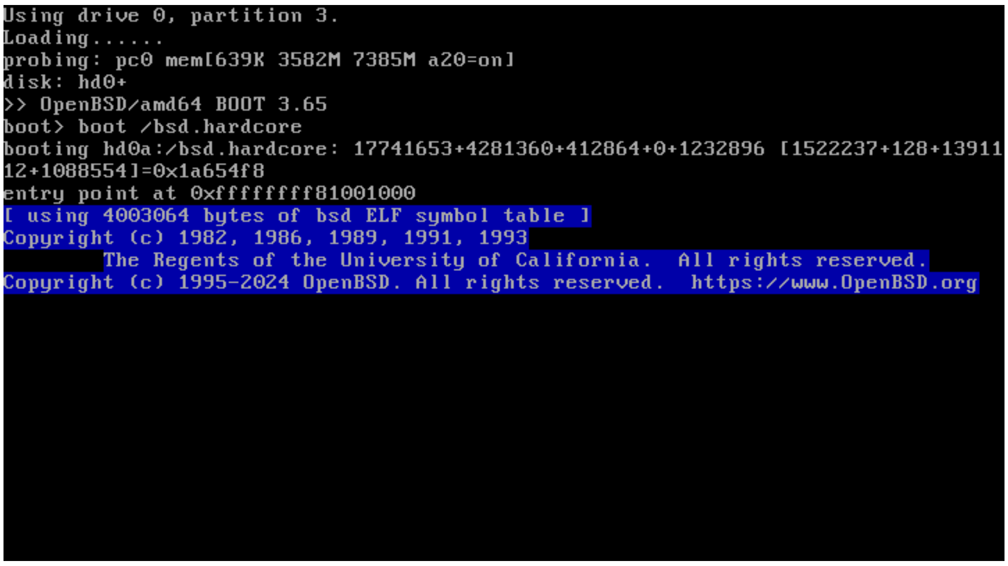
Figure 9: Autopatcher design

Subsystem	**API req**	**Context Tok.**	**Generated Tok.**	**Total Tok.**	**Cost (GPT-4o)**
netinet	301	175241	124913	300154	2.75$
netinet6	565	260905	187643	458548	4.27$

In this test-run, cost was under 10usd for the complete netinet/netinet6 processing, using one of the most expensive models available (GPT-4o). This cost is very small compared to the cost of a developer, but most of the cost of hardening software will be the cost of patch review. Performance of different models regarding autopatching was not measured in this article. Total time spent patching the netinet/netinet6 subsystem was about 12 hs.

4.5 Recommended usage

The autopatcher can generate code with additional checks that may prevent many unknown bugs from being exploited. However, as we can assume that LLMs will continue



```
Using drive 0, partition 3.
Loading......
probing: pc0 mem[639K 3582M 7385M a20=on]
disk: hd0+
>> OpenBSD/amd64 BOOT 3.65
boot> boot /bsd.hardcore
booting hd0a:/bsd.hardcore: 17741653+4281360+412864+0+1232896 [1522237+128+13911
12+1088554]=0x1a654f8
entry point at 0xffffffff81001000
[ using 4003064 bytes of bsd ELF symbol table ]
Copyright (c) 1982, 1986, 1989, 1991, 1993
        The Regents of the University of California.  All rights reserved.
Copyright (c) 1995-2024 OpenBSD. All rights reserved.  https://www.OpenBSD.org
```

Figure 10: OpenBSD 7.5 with AI-hardened IP stack patches booting.

to improve at a fast rate, it is not recommended to commit the generated checks to the code permanently, as they can be easily regenerated when needed with more advanced LLMs, generating better checks. In this way, we can see the autopatcher as a pre-compilation stage for most projects.

5 Conclusion

This article shows that current state-of-the-art LLMs can discover some classes of vulnerabilities on real C/C++ projects, specifically memory corruption bugs. And while they are not advanced enough to verify/exploit them, the AI can easily generate and integrate patches that prevent them. We argue that the risk of auto-exploitation of vulnerabilities from advanced AIs is negated by the auto-patching capabilities of not-so-advanced AIs.

The Crashbench benchmark can be used to measure 'code understanding' of LLMs in contrast to code or text generation.

This opens up the possibility of using LLMs as a tool for automatic patching and automatic refactoring of existing code, adding protections or features with little or no human intervention.

6 Document version

25 Jun 2024 First paper version delivered to Off-By-One 2024 conference

27 Jun 2024 Fixes and typos

References

[1] Crashbench github project, `https://github.com/ortegaalfredo/crashbench`, Alfredo Ortega

[2] Insecure Programming Exercises, `https://github.com/gerasdf/InsecureProgramming`, Gerasdf

[3] Vllm Project, `https://github.com/vllm-project/vllm`, Vllm-project

[4] Aphrodite Engine Project, `https://github.com/PygmalionAI/aphrodite-engine`, PygmalionAI

[5] LMSYS Leaderboard, `https://chat.lmsys.org/?leaderboard`, LMSYS

[6] Autokaker/Autopatcher github project, `https://github.com/ortegaalfredo/autokaker`, Alfredo Ortega

[7] CyberSecEval2: A Wide-Ranging Cybersecurity Evaluation Suite for Large Language Models, Manish Bhatt Et Al., 2024

[8] Project Naptime: Evaluating Offensive Security Capabilities of Large Language Models, `https://googleprojectzero.blogspot.com/2024/06/project-naptime.html`, Project Zero

[9] A Massively Spiffy Yet Delicately Unobtrusive Compression Library, `zlib.net`, Jean-loup Gailly (compression) and Mark Adler (decompression)

[10] Hardened refactored zlib, `http://www.github.com/ortegaalfredo/zlib-hardcored`, Alfredo Ortega

[11] Hardened refactored openbsd, `http://www.github.com/ortegaalfredo/openbsd-hardcore`, Alfredo Ortega

[12] The OpenBSD Project, `http://www.openbsd.org`, The OpenBSD Team